Kera: A Unified Storage and Ingestion Architecture for Efficient Stream Processing

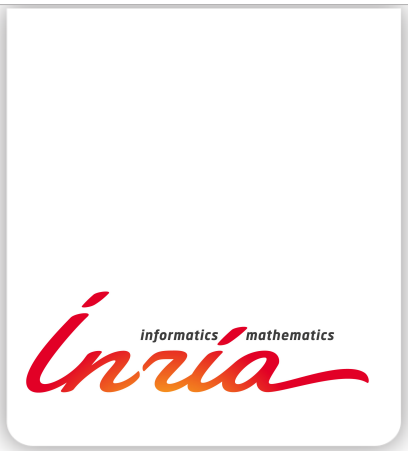
Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María S. Pérez-Hernández

Kera: A Unified Storage and Ingestion Architecture for Efficient Stream Processing

Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María S. Pérez-Hernández

TECHNICAL REPORT
N° 9074
June 2017
Project-Team KerData

ISSN 0249-0803 ISRN INRIA/RT--9074--FR+ENG



Kera: A Unified Storage and Ingestion Architecture for Efficient Stream Processing

Ovidiu-Cristian Marcu*, Alexandru Costan†, Gabriel Antoniu‡,

María S. Pérez-Hernández§

Project-Team KerData

Technical Report n° 9074 — June 2017 — 24 pages

* Inria Rennes - Bretagne Atlantique, {ovidiu-cristian.marcu}@inria.fr
† IRISA / INSA Rennes, KerData, alexandru.costan@irisa.fr
‡ Inria Rennes - Bretagne Atlantique, KerData, gabriel.antoniu@inria.fr
§ Universidad Politecnica de Madrid, mperez@fi.upm.es

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Abstract: Big Data applications are rapidly moving from a batch-oriented execution to a real-time model in order to extract value from the streams of data just as fast as they arrive. Such stream-based applications need to immediately ingest and analyze data and in many use cases combine live (i.e., real-time streams) and archived data in order to extract better insights. Current streaming architectures are designed with distinct components for ingestion (e.g., Kafka) and storage (e.g., HDFS) of stream data. Unfortunately, this separation is becoming an overhead especially when data needs to be archived for later analysis (i.e., near real-time): in such use cases, stream data has to be written twice to disk and may pass twice over high latency networks. Moreover, current ingestion mechanisms offer no support for searching the acquired streams in real time, an important requirement to promptly react to fast data.
In this paper we describe the design of *Kera*: a unified storage and ingestion architecture that could better serve the specific needs of stream processing. We identify a set of design principles for stream-based Big Data processing that guide us in designing a novel architecture for streaming. We design Kera in order to reduce the storage and network utilization significantly, which can lead to reduced times for stream processing and archival. To this end, we propose a set of optimization techniques for handling streams with a log-structured (in memory and on disk) approach. On top of our envisioned architecture we devise the implementation of an efficient interface for data ingestion, processing, and storage (DIPS), an interplay between processing engines and smart storage systems, with the goal to reduce the end-to-end stream processing latency.

Key-words: Big Data, Streaming, Storage, Ingestion, Unified Architecture, End-to-end Stream Processing Latency, Log-structured Memory, Anti-caching, RAMCloud, HDFS, Kafka

1 Introduction

1.1 Current Big Data Trends: Real-time Stream Processing

We are witnessing a rapid change in the mindset of Big Data stakeholders, under the pressure of the Velocity challenge. In more and more application areas data to be processed has a dynamic nature (i.e., streams), and the insights that can be extracted from streams should now be immediately provided, just as fast as the streams are actually ingested into the processing systems. In this context, online and interactive Big Data streaming tools are rapidly evolving as MapReduce [1] is not effective in responding to such critical demands [2, 3, 4].

In order to cope with this new, *online dimension* [5, 6] of data processing, both industry and academia recognized the need to build tools that reason about time: *the world beyond batch processing* is characterized by novel streaming analytic tools like Apache Flink [7], Apache Storm [8], Apache Spark [9], Streamscope [10], Apache Samza [11], or Apache Apex [12], that respond to most requirements of real-time stream processing [13].

1.2 Current Streaming Architectures: Distinct Systems for Ingestion and Storage of Streams

As depicted in **Figure 1**, current streaming architectures are built on top of a three layer stack:

1. **Ingestion** components serve to acquire stream data and eventually pre-process (e.g., filter) it before it is actually consumed by streaming engines. The ingestion layer does not guarantee persistence: it temporarily holds data and provides limited data access semantics based on records (e.g., Apache Flume) or offsets (e.g., Apache Kafka).

2. **Stream processing** engines (e.g., Apache Flink) consume stream elements (i.e., record by record) delivered by the ingestion layer and store the produced results within the underlying storage layer.

3. **Storage** systems are critical components in streaming architectures: they temporarily/permanently store stream data (i.e., stream records) or durably retain processed streams (i.e., partial or final results) that are later queried for other purposes. This layer is typically based on HDFS [14], for interoperability reasons with previous developments in the area of Big Data analytics.

1.3 Identified Architectural Limitations and Processing Overheads

The wisdom brought from the philosophy *'no one size fits all'* forced system designers to develop *dedicated solutions* for both stream data ingestion (e.g., Kafka) and storage (e.g., HDFS). However, these lambda architectures [15] are quite difficult to couple and to maintain, mainly because they were designed as robust and fault-tolerant systems able to serve a wide range of workloads (i.e., support both online and offline data access patterns required by certain streaming use cases). This difficulty is augmented by novel use cases (detailed in the next section) where real-time processed data needs also to be archived for a certain period of time. This archival requirement coupled with real-time processing constraints (e.g., fast ingestion, search primitives on streams) brings unnecessary overheads of storage and network utilization (as we illustrate in Section 4). These limitations and overheads contribute to increased processing costs (i.e., more resources needed) and to an increased end-to-end stream processing latency.

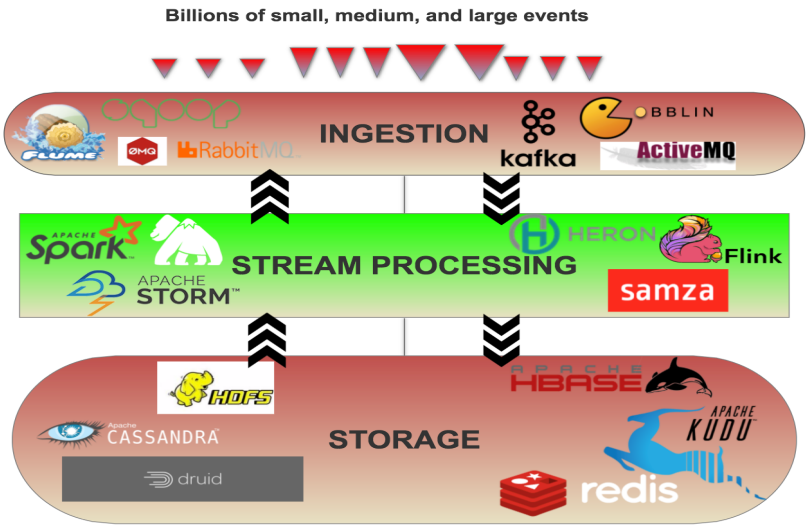


Figure 1: The usual streaming architecture: data is first ingested and then it flows through the processing layer which relies on the storage layer for storing aggregated data or for archiving streams for later usage.

1.4 Proposed Approach and Optimization Objective

In order to 1) circumvent the overhead brought to the processing layer by these specific, yet heterogeneous, hard to configure and optimize building blocks, and to 2) better serve the specific needs of stream processing, in this paper we describe *Kera*: *a unified storage and ingestion architecture* for efficient stream processing. We identify a set of design principles for stream-based Big Data processing that guide us in designing a novel architecture for streaming. Kera is designed in order to reduce the storage and network utilization significantly, which can lead to reduced processing times for stream processing and archival. On top of our envisioned architecture, we devise the implementation of an efficient interface for data ingestion, processing, and storage (DIPS), an interplay between processing engines and smart storage systems, with *the goal to reduce the end-to-end stream processing latency.*

1.5 Paper Structure

We first present a set of motivating use cases (Section 2) that exhibit challenging requirements for fast ingestion and low latency storage support, which are inefficiently handled by current stream architectures. After, we describe the main metric of stream-based processing that we want to optimize: reducing the end-to-end processing latency of stream events is very important to final users (Section 3); this means that we can optimize the processing performance for the same hardware resources (i.e., extract more value) or stream-based applications can use less resources for the same processing needs (i.e., reduced costs). Then, we describe (at high-level) our envisioned approach for a smart stream handling architecture that unifies storage and ingestion, showing a potential reduction in storage and network utilization (Section 4). Next, considering the challenges and requirements discussed in previous sections, we identify and describe a set of design principles for stream-based Big Data processing (Section 5): they will guide and influence our architectural decisions for building Kera.

At this point, we have motivated our work and laid the necessary foundations to describe the architectural elements of our novel architecture for streaming (Section 6). We start by describing the characteristics of our architecture: stream data model, layout, and partitioning (Subsection 6.1). Then, we present the overall architectural design and we describe the flow of a stream's records by looking at the interaction with the ingestion and archival components (Subsection 6.2). Next, we identify and describe the necessary techniques that will optimize the ingestion (Subsection 6.3) and archival (Subsection 6.4) of streams. After, we describe the DIPS interface and the necessary APIs for consuming and producing streams, paying attention to the required metadata, in order to boost an efficient stream processing (Subsection 6.5). Finally, we review the systems that inspired our work and describe some of their interesting properties that strongly support our architecture, but also their limitations that we managed to identify and overcome (Subsection 7). We provide a discussion of our framework (Subsection 8) and we finish by concluding our work and discussing the next steps necessary to validate the concepts and techniques of our proposed architecture (Subsection 9).

2 Motivation: Data Streaming Use Cases

Stream processing can solve a large set of business and scientific problems including network monitoring, real-time fraud detection, e-commerce, etc. Essentially, these applications require real-time processing of stream data (i.e., unbounded sequence of events), in order to gather valuable insights that immediately contribute with results for final users: streams of data are pushed to stream systems and queries are continuously executed over them [16]. We describe below examples of stream-based applications and their specific requirements for ingestion and storage.

2.1 Monetizing Streaming Video Content

This use case is described in [6] to motivate the dataflow model that Google uses for stream processing (i.e., windowing, triggering, and incremental processing). Streaming video providers display video advertisements and are interested in efficiently billing their advertisers. Both video providers and advertisers need statistics about their videos (e.g., how long a video is watched and by which demographic groups); they need this information as fast as possible (i.e., in real-time) in order to optimize their strategy (e.g., adjust advertisers' budgets). We identify a set of requirements associated to these applications:

1. Events are *ingested* as fast as possible and consumed by processing engines that are updating statistics in real-time (requires fine-grained access);
2. Aggregated events and processed streams' results are *stored* for future usage (e.g., offline experiments);
3. Users *interrogate* streams (SQL queries on streams) to validate quality agreements.

2.2 Network Monitoring Systems

This use case is described in [17] to motivate a novel stream archiving system. Network monitoring [18] can serve for management and forensic purposes, in order to track system attacks or locate performance problems. Real-time monitoring involves certain steps:

1. Network packet headers are *ingested, indexed and archived* in real-time;

2. Monitoring systems run *continous queries* on live and archived data to generate alerts when problems are identified;

3. Stream-based workflows require long living pipelines to support real-time analytics.

2.3 Decision Support for Smart Cities Applications

The Internet of Things (IoT [19]) applications collect data from a variety of smart and connected devices producing massive volumes of intermittent data streams. Data from such sensors are referred to as time-series (i.e., a collection of data points with time attributes). In this context, the critical challenge is to use this data while it is in motion. Smart Cities are an important IoT use case, leveraging devices installed in a city in order to improve citizens' life. The main requirements of this use case are:

1. Data from sensors may be initially *ingested and pre-processed before it is delivered* to the streaming engines;

2. Architectures have to support *complex queries on time-series data*;

3. *Time is the critical dimension* for efficient reactive decisions;

4. Massive quantities of data are received over short time intervals;

5. Ingestion components have to support a high frequency of stream event rates.

To sum up, stream-based applications strongly rely on the following features, not well supported by current streaming architectures:

1. *Fast ingestion*, doubled by *simultaneous indexing* (often, through a single pass on data) for real-time processing;

2. *Low-latency storage* with *fine-grained query support* for efficient filtering and aggregation of data records;

3. Storage coping with events accumulating in *large volumes over a short period of time*.

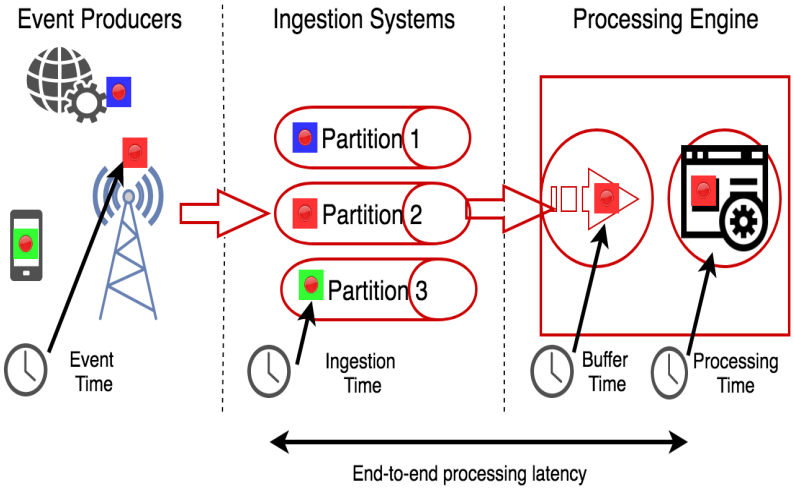


Figure 2: Event producers can be all kind of smart devices, sensors or web services that continuously generate events (i.e., stream records) and are characterized by an *event time*. These events are acquired by ingestion systems, being buffered in stream partitions at *ingestion time*. Processing engines will pull these events from the ingestion system in order to process them in a given setup: the moment these events are buffered at the processing engine is characterized by the *buffer time*, as depicted in the figure. These events are further processed (e.g., by triggering a window-based operator which is slicing a stream in a finite number of events, and executing a user defined function over the window's events and the most recent ingested event) and the result is given at *processing time*, as marked in the figure.

3 Main Metric: Optimizing End-to-end Stream Processing Latency

Users of stream-based Big Data processing are interested in extracting value from their stream data as fast as possible: this is why we look at the main metric in stream processing and try to optimize it. As depicted in **Figure 2**, we define the end-to-end processing latency of an event's record as the time occurred between the (window) processing time (i.e., end of the execution of an event) and the ingestion time. While we cannot control the time spent in order to acquire each event (i.e., ingestion time minus event time, as it depends on the network between the event's producer and the ingestion component in our architecture), **our main goal** is to minimize the time to buffer the record in our processing engine (i.e., the buffer time minus ingestion time).

Also, with an optimized streaming architecture (as proposed in this paper), there is potential to significantly reduce disk and network overheads, which will contribute to the minimization of the end-to-end stream processing latency. Our envisioned architecture could better support the streaming engine for handling stream state, and further reduce processing times. The time spent to process the current event's record (processing time minus buffer time) depends also on other orthogonal mechanisms (e.g., incremental processing, tuple serialization, stream interfaces, etc.) related to the processing engine that we leave for future work.

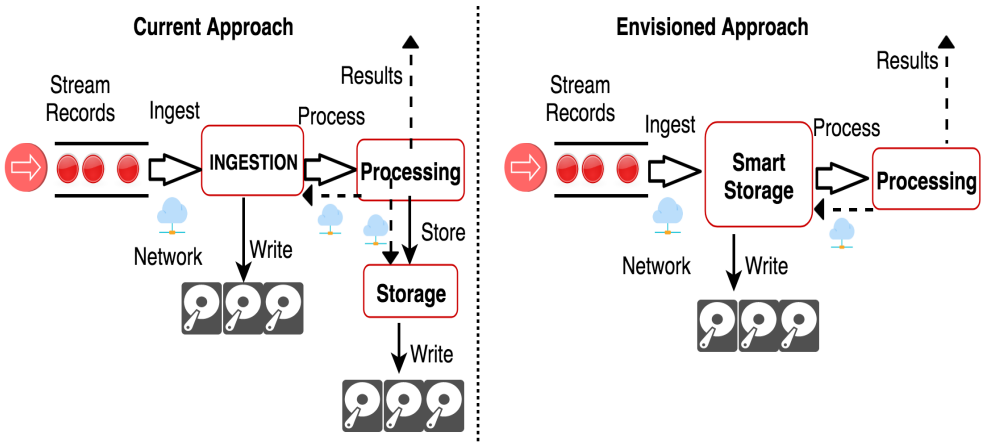


Figure 3: Approaches for handling stream data ingestion and storage. Current approach: streams of records are acquired (the *Ingest* action) by an ingestion component (e.g., Kafka) and records are stored on disk to ensure durability (the *Write* operation); the ingestion component will send acknowledgements to stream producers to certify data was acquired correctly, after which processing engines can pull this data for consumption (the *Process* action). For use cases that require streams to be archived, the processing engine is also responsible to store these streams (the *Store* action) in a storage system (e.g., HDFS), in order to be later queried or analyzed, usually in a batch-oriented manner. Envisioned approach: the Smart Storage architecture includes interfaces for both ingestion and storage; these components cooperate efficiently as in a single system. Stream's results can be pushed back to the ingestion component in order to be consumed by other stream-based applications, or they can be stored in the storage system for other purposes. As a stream's results may need to be archived, having a smart storage with an ingestion interface will allow to immediately archive these results and to avoid aggregation and buffering mechanisms in the processing engine, necessary to efficiently store them.

4 The Need for a Unified Storage and Ingestion Architecture

4.1 The Current Approach: Storage and Network Overheads

Big Data streaming systems rely on message broker solutions that decouple data sources (i.e., data ingestion) from applications (i.e., data processing). As described in **Figure 1**, these architectures typically span over three layers: first to acquire streams, second to process them, and third to store results or to archive streams. However, when deployed in real-life systems, these existing approaches show **important overheads** for use cases where to-be-processed streams need also to be archived for a certain period of time. As detailed in **Figure 3** - the current approach - each record will be written twice to disk (once by the ingestion framework to ensure durability of the acquired records and second by the storage system which simply archive streams for later analysis) and may also be traversing twice the network (the Process and Store actions). With a smart storage approach, the Store action is handled internally, transparent to the processing engine.

4.2 Limitations of a Disk-based Approach for Stream Ingestion

E.g., Apache Kafka - the state of the art system for stream ingestion - writes each record on disk (to ensure data is not lost), then it acknowledges the producer of these records; only after this step Kafka is able to deliver the ingested records to the registered consumers for processing. Kafka is leveraging system's cache for sending these records, however it is not guaranteed that the data to be processed is in cache, as such, for some of a stream's data, **consumers may wait more time due to disk accesses**; this imply that some data may be consumed with lower throughput than serving it directly from memory.

A disk-based design (e.g., Kafka's approach) may serve well to use cases that only need to acquire and store fast streams, that later will moved to a storage system from which can be processed in a batch oriented style. However, for novel use cases that bring challenging access requirements (e.g., fast data access - the need to reduce the access time to stream data such that processing latency is minimized), **we need an approach that guarantees fast data access**: the acquired to-be-processed streams should be consumed directly and immediately from memory (i.e., faster throughput, lower latency), right after producers receive the acknowledgements.

Moreover, current ingestion frameworks offer **no mechanisms to search for records in the acquired streams** (e.g., fine-grained access, range queries, scans). E.g., Kafka offers limited support, being able only to replay parts of a stream, by means of an offset (a number associated to each record). We have to consider such requirements when designing the ingestion layer of a smart storage architecture.

4.3 Envisioned Approach: The Smart Storage

With a unified storage and ingestion architecture, such overheads and limitations (writing twice to disk, possibly waiting for disk by consumers, passing the network for processing and archival etc.) can be overcome. Enhancing storage solutions with ingestion capabilities (i.e., smart storage) will also help, on the one hand, developing complex *stream-based workflows* (i.e., by allowing to pass intermediate/final aggregated stream results to other streaming applications) and, on the other hand, better supporting *stream checkpointing* techniques (i.e., by efficiently storing temporary results).

Moreover, it is not necessary anymore, for the to-be-archived records, to pass the processing layer before they are sent for archiving (the *Store* action is handled internally by the smart storage). Currently, for use cases that need to archive the processed events, the processing engine will aggregate events in blocks and write them to storage in a block-by-block fashion. This means that by transferring the function for archival of stream records to the storage engine, the processing engine will have more time to process more records, while the streams' archival is more efficient. It results that we can optimize the processing performance - higher throughput, lower end-to-end latency (intuitively, less time the record will be buffered at the processing engine for archival purposes, less impact the queueing effect will have on the end-to-end processing latency of unprocessed records). We also observe that in order to enforce harder latency bounds (e.g., a 99% SLA) on event processing latency, it is easier to manage it having an unified storage and ingestion architecture (events are ingested, processed, and stored potentially on the same node, avoiding network transfers).

5 Design Principles for Stream-based Big Data Processing

In building an unified storage and ingestion architecture for stream-based processing, we retain a set of design principles that will guide and influence Kera's architectural decisions.

1. **Processing Engines Should Model Computation**. They should focus on how to transform data. Processing engines should offer the necessary API and semantics for user-defined computation flows and optimizations of the execution flow. Processing engines should follow a dataflow graph-like execution: a natural choice for handling streams of records (subsuming batch execution). They should avoid complicated mechanisms to handle state at this level: it is more efficient to leave this function to specialized (i.e., smart) storage engines such as the one we envision.

2. **Storage Systems Should Model Data Movement**. They should handle the ingestion and storage of stream records (two different functions that are also complementary). Their role is to represent data: how it is stored (persisted in memory or on disk) and how it is partitioned. Storage systems should represent data state through an interface capable of fast ingestion of streams of records, storing data and providing efficient access for stream data to the processing engines.

3. **A Common Abstraction for the Interaction between Processing Engines and Storage Systems Should Be Designed**. Let's name it DIPS (i.e., to handle data ingestion, processing, and storage for streams of records). Storage systems and processing engines will understand the same interface (DIPS). DIPS represents streams of records and offers APIs to read and write a stream. DIPS will leverage data immutability and sequential access, natural characteristics of streams. DIPS is flexible for optimizations: smart storage (when storage systems are in control) and efficient processing (when processing engines are the drivers of the execution).

4. **Memory Is the New Disk: the Anti-caching Principle**. In contrast to current strategies (writing streams to disk, e.g. by a disk-based ingestion framework like Kafka, before they are delivered to a processing engine), stream data should be available for processing, right after we have acquired it, that is from memory, and should not wait for disk. This means we should adopt the *anti-caching* principle [20] when processing streams, a recent trend in building novel databases.

5. **Leverage Log-structured Storage in Memory and on Disk**. This is a decision that comes naturally, being emphasized by the structure of a stream: data arrives in a record-by-record fashion and it is processed and archived similarly. Moreover, leveraging sequential access to streams, in memory or on disk, will maximize performance (i.e., throughput, latency).

6. **Model Stream Records with a Multi-key-value Format**. Stream records are traditionally modelled with a simple key-value format, where value is an uninterpreted blob of data (e.g., in Kafka). In order to have secondary indexes, required to efficiently search for records by their attributes (including the primary key), clients and servers have to agree on where the secondary keys are located in the record. To efficiently index a stream's records, we want to avoid parsing the record's value. A record's structure will contain a primary key, optionally multiple secondary keys, and a variable-length uninterpreted value blob (allows for other attributes, e.g., like in [21]), in order to give more flexibility to an enhanced storage-ingestion architecture.

7. **Enable Fast Crash Recovery Techniques for Handling Fault-Tolerant Streams** Fault-tolerant (storage) systems are able to continuously perform their function in spite of errors. Users of stream-based applications require low latency answers in any conditions. In order to guarantee such strict requirements, we need to employ techniques that

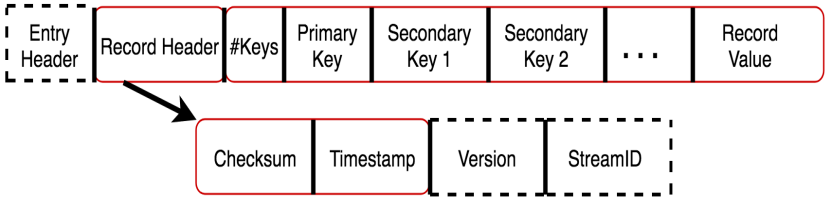


Figure 4: Multi-key-value Record format for streams: Version, StreamID, and EntryHeader are not necessary for our immutable uniform segments; for mutable streams with non-unique records we retain Version and EntryHeader.

are recognized to recover data as fast as possible. One such technique is the fast crash recovery mechanism implemented in RAMCloud [22], which is leveraging the aggregated disk bandwidth in order to recover the data of a lost node in seconds.

6 Kera: Architectural Elements

In this section we describe the general architecture of Kera, explaining in detail the ingestion and storage components and their interactions, and after we present the DIPS interface for data ingestion, processing, and storage that we devise on top of Kera.

We first define and explain the stream data characteristics of our unified architecture for streaming: how streams are modelled, partitioned and the layout of stream's records in memory and on disk.

6.1 Stream Data Characteristics for Ingestion and Storage

Stream Definition. A stream is an unbounded sequence of events that needs (near) real-time processing and has (possibly) multiple consumers and/or producers. Events are not necessarily correlated to each other and each event has one record that has to be ingested, processed, and stored, just as fast as it arrives.

6.1.1 Stream Data Model.

Each stream is modeled as a *log*. Each log contains a possibly infinite number of *segments*. Each segment has a fixed size (e.g., 8MB) and contains a number of records. A *record* has a multi-key-value format, similar to RAMCloud's records: we give a representation of stream's records in **Figure 4**. We discuss the record's metadata overhead in the next section.

6.1.2 Stream Data Layout.

Segments have the same structure on both disk and memory. Each segment will contain records from a single stream (we call these *uniform segments*). This is in contrast to log-structured key-value stores, such as Ramcloud, which ingest records from multiple streams in the same segment (unique at a time) called the head segment of the managed log. Uniform segments are necessary in order to efficiently acquire, consume and archive streams. We want to avoid for a log's segment to contain records from different streams (some applications may need to consume only certain streams; consuming records from a single stream is also a requirement of stream-based applications): uniform segments will ensure sequential access, an important optimization.

Unprocessed segments should stay in memory (live segments) and processed segments will be stored on disk (archived, old segments) or simply evicted from memory. A configurable number of segments of the same stream (e.g., each N=16 segments) are grouped into a block and archived as such on disk. Archived segments/blocks will have associated specific metadata records in order to be easily identified by indexes, based on the following parameters: [streamId, blockId, segmentId].

6.1.3 Stream Partitioning.

Each stream can be seen as a log of records, each log is partitioned in *streamlets* (based on hashing intervals), each streamlet contains a set of *uniform/live segments* (i.e., these segments are stored in memory) and possibly a set of archived segments. A master will manage a set of streamlets for a set of streams. Consumers can process segments of a streamlet based on consistent hashing techniques.

6.2 Design of a Unified Storage and Ingestion Architecture

First, we present the overal architectural design, then we describe the general flow of a stream's records interactions with respect to the ingestion and archival components.

6.2.1 General Architecture.

As depicted in **Figure 5**, we have three components: a *Coordinator* that is managing the aggregated log- structured memory of a set of nodes where streams are ingested, a *NameNode* responsible to manage the archival of streams on disk, and a *Processing Engine Master* that is managing the Workers responsible to process the acquired streams. Each component can scale independently. To maintain high availability, component's data structures can be stored in a system like ZooKeeper, a replicated, highly-available configuration storage system (also, each component will have passive copies that maintain the same state as the active one). The interactions of the stream clients with the main components is reduced, in order for the system to easily scale to thousands of nodes.

The key insight is that recent ingested streams are kept in memory in order to be immediately processed, after which these streams can be optionally archived and evicted from memory. The ingestion and storage components are designed separately for scalability reasons, but they share the resources of a data node. Our design in which we configure a node to be shared between ingestion, processing, and storage components is supported by recent hardware advancements, with trends of increasing the number of cores per node (multi-core, many-core).

6.2.2 Stream Ingestion.

On each data node (server) will live component instances that will manage the ingestion, archival, and processing of the acquired streams. The ingestion component (i.e., has capabilities to ingest records) is called *Master*. A Master will handle one log. Each log will be represented in memory by a set of uniform segments. Segments that are not fully occupied are called *active segments*. An active segment can accumulate new records that are appended to it. Each stream will have its own active segment: this means that at any moment, a Master handling records from multiple streams will manage multiple active segments. We describe in **Figure 6** the process of ingestion of a single stream's records, and we note that it is similarly done for multiple streams, as depicted in **Figure 7**.

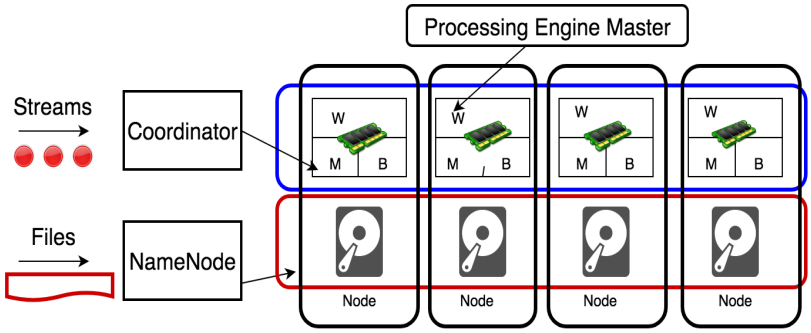


Figure 5: Kera's architecture supports the ingestion of both streams and large files. Stream clients are communicating initially with the coordinator in order to get information about the nodes where produced streams can be ingested. The coordinator will manage a part of the memory of a set of data nodes. Each data node will have a Master instance (depicted M in figure) and a Backup instance (depicted B in figure). Each Master is handling a log and is responsible to ingest and keep a copy of the stream's records. A Master is communicating with a set of Backup nodes in order to replicate acquired records. Disk-based storage is handled by a NameNode, responsible to keep the metadata associated to archived files and streams, and communicate with a set of Data Nodes instances - a Data Node instance is running on each node. We describe later the interactions between Backup instances and the NameNode. Stream processing engines will be coordinated by a Processing Engine Master which is communicating with its Workers (depicted W in figure). Workers and Masters should be collocated on the same node in order to leverage data locality.

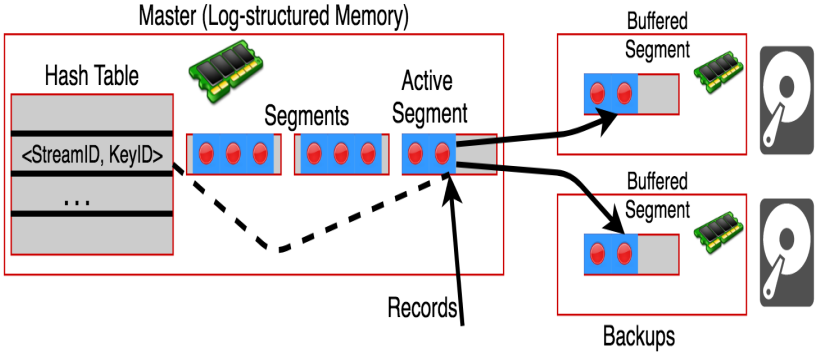


Figure 6: A master server consists primarily of a hash table and an in memory log. The hash table will represent the indexed records. Each new record of a stream is appended to the log's active segment associated to the record's stream and is synchronously replicated to volatile (i.e., DRAM) buffers on backups. Client writes are acknowledged once all backups have buffered the new addition to the active segment. The log is replicated across several backups' disks for durability. Group of full segments (e.g., a group of N=16 segments, each of 8MB) are flushed from the corresponding buffers to disk by an archival component. This operation is independent to the ingestion phase.

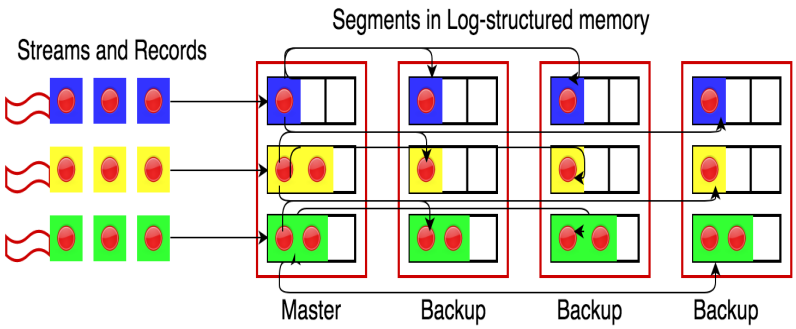


Figure 7: Ingestion of stream records. We represent the ingestion of three streams, with blue, yellow, and green. Records are ingested by a Master and replicated synchronously in memory of three Backups. Each Stream has a **unique active segment** in which its records are appended. To exemplify, the blue stream has a record ingested and replicated, after which an acknowledgement is sent and its clients can start sending another record. The yellow stream is in the process of ingesting a second record, as we can see at the Master, it is not yet replicated on Backups. The green stream has two records ingested and acknowledged and its client can start streaming the third record.

6.2.3 Stream Archival.

Processed streams may need to be permanently archived (streams are persisted to disk for durability, however processed streams may be simply evicted from memory if users do not specify a stream's requirement for archival). We archive the corresponding stream's (active) segments from a master's log. When we archive segments, we group a fixed number of segments (e.g., N=16) and write them as a contiguous *block*. Streams on disk are represented logically as a file, each file is a set of blocks (e.g., like in HDFS). We describe in **Figure 8** the process of archival of a single stream's segments. This process is managed by an archival component, which can be optionally activated (e.g., per stream), and that will write blocks in asynchronous way, independent of ingestion and processing of streams.

6.2.4 Memory Management: Record's Metadata Overhead

As mentioned in **Figure 4**, each record has a multi-key-value format and its representation includes an object header with an overhead of 26 bytes and an additional 2-5 bytes entry header (this is the original representation of a record in RAMCloud; each entry header has an entry type of 1 byte and the entry length of 1 to 4 bytes; an object header has a 64-bit tableID - equivalent streamId, a 64-bit version, a 32-bit timestamp, a 32-bit checksum, and a 16-bit keyLength).

As a uniform segment contains records from a single stream, the 64-bit streamId is not necessary. If a segments' records are immutable and unique, the 1-byte entry type and the 64-bit version are not necessary (they are used by the log cleaner). This means that the total overhead carried by unique, immutable, and uniform segments is 17 bytes, but we require a segment header in which to save the 64-bit streamID in order to identify its records. However, for a stream that has multiple records with the same key, we still need to keep the 64-bit version and 1 byte entry type; in this more general case the overhead is reduced to 8 bytes.

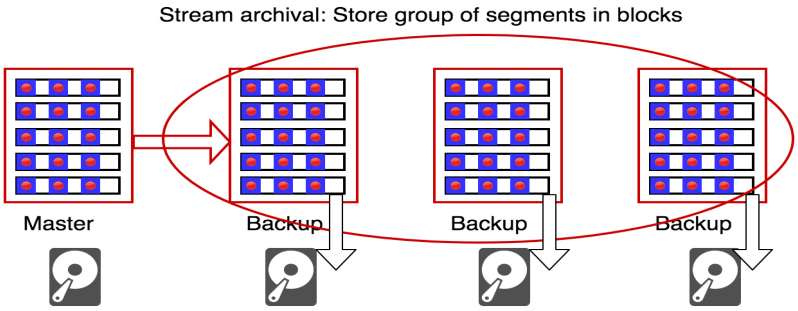


Figure 8: Stream's segments archival required for durability and later reply. We exemplify the archival of a single stream's records. On the Master we have reached the number of segments that was configured for to-be-archived groups (in this case, N=5), and the archival action is triggered. The segments, part of the same group, are replicated on the same Backups. At this moment, the archival component will trigger the archiving process, telling Backups that a block containing the group's segments can be written to disk. A block (the group of segments) is written to disk by the Master and associated Backups. Only Backups will evict the block's segments from memory. We note that the archival operation is done without network usage, as data nodes share the ingestion component. For the next group of segments, another set of Backups will be selected, in order to randomly scatter a stream's segments, a recognized technique necessary for fast crash recovery.

For an immutable stream with unique small records (e.g., 100 bytes) this overhead represents 8% of the used memory (for larger records -e.g., 500-1000 bytes, this overhead is smaller 1-2%). In order to eliminate this overhead, uniform segments should have records with a simpified structure (just as described before). This means that in our architecture we need to manage two types of segments: uniform segments as described previously, and non-uniform segments as RAMCloud is currently managing (in which records are associated to many distinct, mutable streams, possibly smaller). This means we should make RAMCloud (e.g., the log cleaner, the crash recovery mechanism, the replica manager) aware of two types of segments. This optimization is important for immutable streams with (unique) small records (100 bytes) and should be considered in a future version of the prototype.

6.3 Ingestion and Archival Optimizations: Two Log-structured Techniques

The stream ingestion component is using a log-structured in memory design similar to Ramcloud: acquired data is available for processing right after ingestion, but from in memory *uniform segments*. Although our approach resembles Ramcloud's design, there are **two important differences** (we describe them below) that are given by different techniques necessary to optimize the ingestion, processing, and archival of streams. We leverage Ramcloud's repository and modify it accordingly to our proposed techniques, in order to correctly integrate it in our architecture.

6.3.1 First Technique: Active Uniform Segments.

We described in **Figures 6 and 7** that each stream will have its own active segment in which records of the acquired stream are appended. Our strategy is in contrast to Ramcloud's design

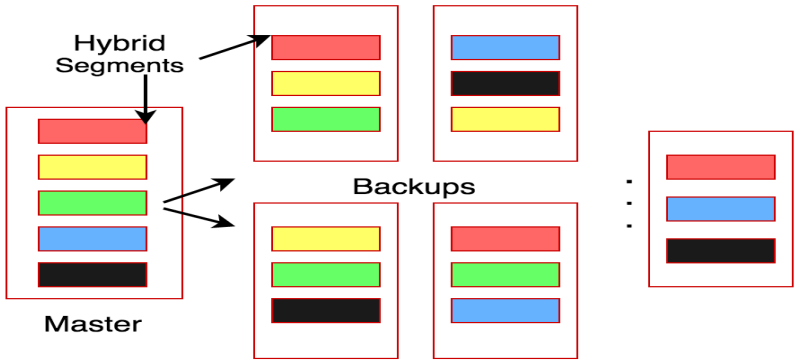


Figure 9: Current strategy in RAMCloud to randomly scatter segments. Hybrid segments contain records from multiple streams. These strategies are not suitable for efficiently handling ingestion, processing and archival of streams.

where each data node (also called a Master) is handling a log in which at any moment there is a single head segment for all streams. This implies that records from different streams will be mixed in the same segment. We want to avoid this situation for two reasons. First, we want the processing engine to efficiently access and consume the stream's records. With our strategy we avoid the overhead of checking if records of a segment are part of the accessed stream. Moreover, our choice of handling unique segments associated to a stream will allow to leverage the sequential access to streams, an important optimization. Second, the same reasoning will allow for an efficient stream archival.

6.3.2 Second Technique: Replicate Group of Segments Similarly.

We described in **Figure 8** how we group segments and replicate each group in a similar way on the same set of Backup nodes (for each group - e.g., N=16 segments - we select a different set of Backups where we replicate a group's segments). Our strategy is in contrast to Ramcloud's design where Master's segments are scattered randomly on a set of backups (this can be visualized in **Figure 9**), a good technique utilized for fast crash recovery, but one which is inappropriate for our archival strategy. Controlling how group of segments are replicated will avoid costly search strategies that could aim to identify such groups in order to efficiently archive them.

6.3.3 Other Considerations.

An important question is related to what to keep in DRAM. We think we should keep in DRAM only to-be-processed streams in order to guarantee an efficient access to consumers. Data will be persisted to disk in order to ensure durability (used by the crash recovery mechanism), but for processed streams not requiring archiving, we can simply evict their segments from memory. As mentioned, data is archived asynchronously by an optional archival component. It will be interesting to see the implications of removing a segment from memory on the RAMCloud's cleaner. As data is immutable and segments are uniform, we think our approach will reduce significantly the work done by the log's cleaner.

6.4 HDFS Extension: A New Block Placement Strategy

We want to leverage HDFS in our architecture when archiving streams and do it with minimal effort. As described before, the key insight is that data (a block is a group of segments) is already replicated in memory on a set of data nodes. The approach we consider is to write these blocks right away from memory to disk, and avoid additional network overhead (**Figure 8**). Archived streams will represent a file in HDFS, and a group of to-be-archived segments will represent a block in HDFS.

We need to define a new *block placement strategy* in HDFS so that a block is persisted directly from memory to disk, avoiding network transfers. To explain how this strategy will optimize the archival process, consider the current approach: stream engines are aggregating stream records in order to archive them to HDFS; current strategy in HDFS when writing a block is to put the block on the first node and in parallel replicate segments of this block on other nodes, traversing the network in order to have multiple copies. This strategy is inefficient: in our case, ingested data is already replicated in memory on a set of data nodes.

We chose HDFS as it is the de facto standard for Big Data analytics storage, being well integrated with the main Big Data processing engines (Apache Spark, Apache Flink). However, other distributed file systems may be considered, with the requirement to support the segment's replication strategy of our ingestion component.

6.5 The DIPS Interface: Required Stream Metadata for Efficient Processing

On top of our architecture we need to develop interfaces required to efficiently produce and consume streams. We differentiate between two categories for stream processing: (1) Producers are responsible to generate/produce streams of records; (2) Consumers are responsible to consume/process streams of records. Producers and consumers of streams will interact through the DIPS interface. At the same time, clients can leverage existing APIs for handling single records and files (e.g., for random access, users can leverage RAMCloud's API write and get, for scans and range queries, users can leverage RAMCloud's secondary indexes, for handling files users can leverage HDFS's file operations).

DIPS will manage log's segments, a segment representing the DIPS primitive and the segment id (SID) will be part of managed stream metadata. Two stream APIs will be exposed: *writeStream* for producers and *readStream* for consumers. Internally, each API may leverage existing RAMCloud's API multiwrite and multiread.

6.5.1 Management of Producers.

A producer will write a stream's records in a Master's log, which will be represented by a set of live (to be processed) and archived (processed) segments. When multiple producers are writing records that are part of the same stream, we leverage the uniform segment technique, and rely on current mechanisms in RAMCloud needed to ensure replication and the order of records in replicated segments (this is explained in **Figure 10**).

What will be interesting is to configure a quota on how much memory space a stream can rely on. This means that some segments of the acquired stream, not yet processed, could be evicted from memory to disk, in order to make room for other records, even if these archived segments are not yet processed. This split between live and old segments should be handled by DIPS with specific metadata [streamId, blockId, segmentId, processedFlag].

In the initial phase of our prototype, we consider there is enough memory space to ingest streams while they are processed. This means that the speed of producing records is not over-

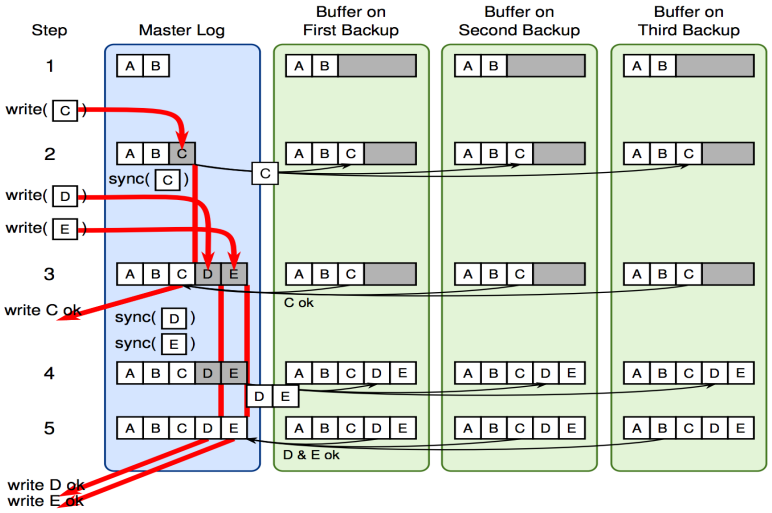


Figure 10: Maintaining the order of ingested records in consistent, uniform segments. Step 1: Records A and B are replicated on backups in memory. Than, a producer asks to write record C. Step 2: During the time a Master is replicating C on its backups, new requests to write records D and E arrived. Step 3: C is acknowledged, while D and E are copied to the active uniform segment. Step 4: D and E are batched in a single replication operation. Step 5: Producers of D and E are acknowledged [figure taken from RAMCloud]

coming the speed of consuming them. If needed to ingest more records than what the current configuration can consume, we can add other nodes to make room for new ingestion and processing capabilities. Else, we throttle the ingestion capabilities as it makes no sense to ingest more than what can be processed.

At this stage, the *writeStream* API will simply wrap the *multiwrite* mechanisms in RAMCloud, while DIPS will internally manage the produced streams, by storing metadata about live and archived segments of a stream.

For future extensions of the prototype, we should build a mechanism to ensure memory quotas on streams, and manage the following important situations: the speed of producing/consuming records may be faster than that of consumers/producers, the available memory for unprocessed streams may dry faster than anticipated (these issues can be solved with current routing mechanisms already developed by specialized ingestion frameworks; the idea is to dynamically change the way records are routed to data nodes).

6.5.2 Management of Consumers.

A consumer will specify a *stream offset* in order to consume a stream's records. This offset is a stream metadata that represents the last record consumed in a stream ([StreamId, SegmentId, Record]). Consumers will pull or push records from segments, one by one or in batches - this is in order to ensure tradeoffs between latency and throughput. It will be interesting to verify if a *push* strategy can create backpressure issues, as processing and ingestion components share the same data node, buffering may not be needed at processing side, being already handled by the ingestion component. So a consumer just triggers the processing of a stream, setting an offset, and the ingestion component can push records in a segment by segment fashion, being notified

by the consumer at a segment granularity. Strategies to acknowledge what has been consumed can be: on a record by record basis, a number of records, or one segment, by means of the last offset defined.

Multiple consumers may want to access the same stream. In this case, a certain technique is necessary to tag a stream's segments in order to decide what was processed by all consumers and what is still required. This should be handled internally by DIPS. Consumers rely on the *readStream* API, but will require other internal APIs, managed by DIPS: *getLiveActiveSegments and getArchivedSegments* of a stream needed to identify the live/old segments, *streamSegments* needed by DIPS to know from where to pull or push records from/to, *streamRecordsOfSegments* to pull or push records by/to consumers, *updateOffset* in order to know what parts of a stream were processed by a consumer.

One challenge is to decide how to efficiently and consistently handle producers and consumers metadata. We think that the best approach is to rely on the features exposed by the ingestion component. DIPS clients can create special 'metadata' streams where metadata information records (e.g., as defined previously) required by producers and consumers can be acquired and queried as needed.

7 Strengths and Limitations of State of Art Ingestion and Storage Systems

In this section we review the systems that inspired our work and we describe their interesting properties that strongly support our architecture, but also their limitations that we managed to identify and overcome.

RAMCloud [23] is an in memory key-value store that aims for low latency reads and writes, by leveraging high performance Infiniband-like networks. Durability and availability are guaranteed by replicating data to remote disks. Among its features we note fast crash recovery [22], efficient memory usage [24] and strong consistency. Recently, RAMCloud was enhanced with multiple secondary indexes for each table [21, 25], achieving high availability by distributing indexes independently from their objects (independent partitioning). Its current implementation for storing records (one active segment per master) makes difficult to directly couple it in our architecture (as we explained in Subsection 6.3). Our changes imply that each stream is associated with its own active segment, and may add a small overhead to the ingestion of key-value records (although solutions to cache this mapping, for tens of ingested streams, will alleviate this overhead). Also, our strategy of scattering segments may imply a less optimized crash recovery, however we think that this is a fair price to pay in order to leverage Ramcloud's functionalities and build an efficient stream ingestion and archival component.

The Hadoop Distributed File System (HDFS) [14] provides scalable and reliable data storage (sharing many concepts with GoogleFS[26]), and is recognized as the de facto standard for Big Data analytics storage. Although HDFS was not designed with streams in mind, many streaming engines depend on it. Among HDFS's limitations we have the metadata server (called NameNode): this is a single point of failure and a source of limited scalability. Also, there is no support for random writes as a consequence of the way the data written is available for readers (data can only be appended to files, only after the file is closed clients have access to data). In fact, in [27] authors point out that HDFS does not perform well for managing a large number of small files, and discuss certain optimizations for improving storage and access efficiencies of small files on HDFS. This is why streaming engines develop custom solutions to overcome HDFS limitations. We enable HDFS in our architecture by designing a new block placement strategy

which will efficiently persist blocks and their replicas, directly from memory, avoiding network costs that already were paid by the replication mechanism of the ingestion component.

Apache Kafka [28] is a distributed stream platform that provides availability and publish/-subscribe functionality for data streams (making producers' stream data available to multiple consumers), being the de facto open source solution (for temporal data persistence and availability) used in end-to-end pipelines with state-of-the-art stream engines like Apache Spark or Apache Flink, which at their turn provide data movement and computation. However, in Kafka there are no means to search for some records (e.g., scans, range queries, random access) in the ingested streams. Kafka's persistence model is disk-based and consuming ingested streams may require access to disk, reducing throughput. Kafka's data model (i.e., key-value) is simple and may impose difficulties (parsing an uninterpreted blob) in use cases that require secondary indexes on various attributes in order to efficiently expose search primitives. We consider Kafka a good solution for use cases that need only to ingest and archive data streams, and only later to be processed in a batch fashion style (these use cases do not require end-to-end low latency stream processing). We explained that using Kafka for real-time processing may not be efficient due to its disk-based architecture.

We argued for an in memory, log-structured approach for stream processing and we designed a set of techniques for efficient ingestion and archival of streams, needed to alleviate the main limitations of RAMCloud, and to easily integrate HDFS. Although RAMCloud, the state-of-art key-value architecture for low latency processing, brings powerful features required also by novel stream-based applications, it was designed without considering one natural evolution of Big Data processing: real-time stream processing.

8 Discussion

As previously mentioned, our main goal is to reduce the end-to-end stream processing latency, and we focus on two requirements of stream-based applications: first is to ingest stream data very fast and second is to archive efficiently to-be-processed streams in order to have search primitives (e.g., scans, range queries, random access) on the live/archived streams. We also remember that our stream architecture should integrate the capabilities of a distributed file system, as some use cases require access to large files in a batch manner. We design an architecture that supports these requirements and we describe the necessary optimization techniques for ingestion, processing, and archival that will contribute to a reduced end-to-end stream processing latency.

First, we design the ingestion capabilities by considering a log-structured approach, with data ingested directly in memory. In order to have fine-grained access to ingested records (put, get key), we have identified a key-value store (i.e., RAMCloud) which responds to our requirements. However, simply integrating RAMCloud in our architecture does not work. Each stream requires its own active segment in which is writing records, in contrast to Ramcloud's approach of keeping a single active segment for all streams. This optimization technique will allow us to leverage the sequential access to a stream's data. Also, we want to efficiently archive segments on disk and avoid another step of replication. Our insight is that we should group segments and replicate them similarly. This is in contrast to Ramcloud's strategy of scattering segments randomly. With two simple techniques we manage to adapt Ramcloud to support our streaming architecture, while keeping its powerful features.

Second, we looked at integrating the disk-based storage component, and we consider HDFS the main choice. However, simply integrating HDFS in our architecture is not efficient: the key insight in our architecture is that ingested streams are already replicated in memory. So writing to disk can be coordinated so that we avoid network usage, as normally done when a file's blocks

are written to HDFS. We recognize that we need a new block placement strategy, one in which we inform the NameNode where are the blocks and exactly where we want to write our blocks and their copies. With a simple technique we manage to extend HDFS in order to support billions of records, and in the same time have access to its features.

Third, we describe an interface (DIPS) for consuming and producing streams on top of our architecture. We acknowledge that DIPS is necessary to efficiently leverage our architecture: although we ingest and consume streams in a record-by-record fashion, we need the optimized mechanisms for trading low latency with high throughput, and this is realized through DIPS. DIPS will take the burden from stream processing engines of being always struggling to find techniques for managing the stream state. DIPS will allow to push computation at the storage level, techniques that researchers show to be important in order to obtain big improvements in processing times [29].

9 Conclusion and Next Steps

We presented the design of a unified storage and ingestion architecture (i.e., the smart storage Kera) and we explained the necessary techniques (i.e., manage uniform active segments for each stream, strategy to randomly scatter groups of segments, i.e., blocks, provide a block placement strategy for efficient archival) needed to optimize the ingestion and archival of streams. Than, we argued for the need of a smart interface to ingest, process, and store streams (i.e., DIPS) and we described a set of APIs needed to consume and produce streams on top of a log-structured ingestion component.

We carefully considered the interactions between ingestion, processing, and storage components, aiming to significantly reduce the disk and network usage, with the main goal of reducing the end-to-end stream processing latency. The main design approach that we consider, in contrast to current ingestion techniques, is the anti-caching principle: 'memory is the new storage', while disk is used for backup and archival of streams [30]. The key observation that we retain is that data is ingested and kept in memory until after processing, when we think is 'safe' (for processing performance) to write it to disk if archival requirements are needed.

Our architecture, having the best of two worlds - fine-grained access and large file support - and being designed for real-time, low latency stream processing, will help users reduce the costs of their infrastructures, while bringing new opportunities for draining more value from their stream data, due to reduced end-to-end stream processing latencies.

The next steps are the implementation of the presented techniques and the validation of our proposed prototype. To do so, the first challenge is to efficiently handle multiple uniform segments by a single master and its backups. Than, the second challenge is to minimize the effect of the segments' group replication on crash recovery, while ensuring an efficient stream processing.

Acknowledgment

This research was sponsored by Huawei HIRP OPEN program. This work is part of the BigStorage [31] project, funded by the European Union under the Marie Sklodowska-Curie Actions (H2020-MSCA-ITN-2014-642963).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on OSDI*. USENIX Association, 2004.

[2] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators," in *ICDT*. Springer-Verlag, 2005, pp. 37–52.

[3] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[4] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at facebook," in *ACM SIGMOD*, 2016, pp. 1087–1098.

[5] G. Fox, S. Jha, and L. Ramakrishnan, *STREAM2016: Streaming Requirements, Experience, Applications and Middleware Workshop*. United States: SciTech Connect, Oct 2016.

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, Aug. 2015.

[7] "Apache Flink." http://flink.apache.org.

[8] "Apache Storm." http://storm.apache.org.

[9] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM SOSP*, 2013, pp. 423–438.

[10] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "Streamscope: Continuous reliable distributed processing of big data streams," in *Usenix NSDI*, 2016, pp. 439–453.

[11] "Apache Samza." http://samza.apache.org/.

[12] "Apache Apex." http://apex.apache.org/.

[13] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, Dec. 2005.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *IEEE MSST*, 2010, pp. 1–10.

[15] M. John, A. Cansu, Z. Stan, T. Nesime, and D. Jiang, "Data ingestion for the connected world," in *CIDR, Online Proceedings*, 2017.

[16] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *IEEE ICDE*, 2005, pp. 779–790.

[17] P. J. Desnoyers and P. Shenoy, "Hyperion: High volume stream archival for retrospective querying," in *USENIX ATC*, 2007, pp. 4:1–4:14.

[18] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *ACM SIGMOD*, 2003, pp. 647–651.

[19] M. Stolpe, "The internet of things: Opportunities and challenges for distributed data analysis," *SIGKDD Explor. Newsl.*, vol. 18, no. 1, pp. 15–34, Aug. 2016.

[20] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik, "Anti-caching: A new approach to database management system architecture," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1942–1953, Sep. 2013. [Online]. Available: http://dx.doi.org/10.14778/2556549.2556575

[21] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout, "Slik: Scalable low-latency indexes for a key-value store," in *USENIX ATC*. Berkeley, CA, USA: USENIX Association, 2016, pp. 57–70.

[22] "Durability and crash recovery in distributed in-memory storage systems," http://web.stanford.edu/~ouster/cgi-bin/papers/StutsmanPhd.pdf.

[23] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2806887

[24] "Memory and object management in ramcloud," http://web.stanford.edu/~ouster/cgi-bin/papers/RumblePhd.pdf.

[25] "Scalable low-latency indexes for a key-value store," https://web.stanford.edu/~ouster/cgi-bin/papers/KejriwalPhd.pdf.

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SOSP*. New York, NY, USA: ACM, 2003, pp. 29–43.

[27] B. Dong, Q. Zheng, F. Tian, K.-M. Chao, R. Ma, and R. Anane, "An optimized approach for storing and accessing small files on cloud storage," *J. Netw. Comput. Appl.*, vol. 35, no. 6, pp. 1847–1862, Nov. 2012.

[28] "Apache Kafka." https://kafka.apache.org/.

[29] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, "Bridging the gap between hpc and big data frameworks," *Proc. VLDB Endow.*, vol. 10, no. 8, pp. 901–912, Apr. 2017. [Online]. Available: https://doi.org/10.14778/3090163.3090168

[30] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: Scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1713254.1713276

[31] "Bigstorage." http://bigstorage-project.eu/.

Contents

1 Introduction **3**
1.1 Current Big Data Trends: Real-time Stream Processing . . . 3
1.2 Current Streaming Architectures: Distinct Systems for Ingestion and Storage of Streams . . . 3
1.3 Identified Architectural Limitations and Processing Overheads . . . 3
1.4 Proposed Approach and Optimization Objective . . . 4
1.5 Paper Structure . . . 4

2 Motivation: Data Streaming Use Cases **5**
2.1 Monetizing Streaming Video Content . . . 5
2.2 Network Monitoring Systems . . . 5
2.3 Decision Support for Smart Cities Applications . . . 6

3 Main Metric: Optimizing End-to-end Stream Processing Latency **7**

4 The Need for a Unified Storage and Ingestion Architecture **8**
4.1 The Current Approach: Storage and Network Overheads . . . 8
4.2 Limitations of a Disk-based Approach for Stream Ingestion . . . 9
4.3 Envisioned Approach: The Smart Storage . . . 9

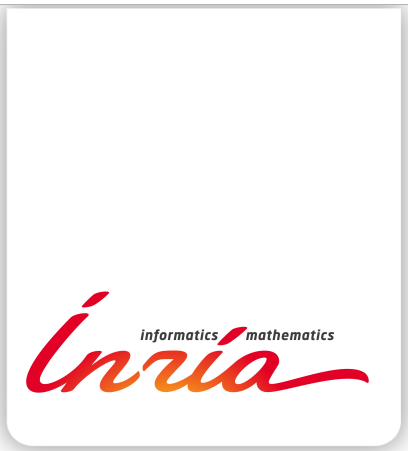
5 Design Principles for Stream-based Big Data Processing **9**

6 Kera: Architectural Elements **11**
6.1 Stream Data Characteristics for Ingestion and Storage . . . 11
6.1.1 Stream Data Model. . . . 11
6.1.2 Stream Data Layout. . . . 11
6.1.3 Stream Partitioning. . . . 12
6.2 Design of a Unified Storage and Ingestion Architecture . . . 12
6.2.1 General Architecture. . . . 12
6.2.2 Stream Ingestion. . . . 12
6.2.3 Stream Archival. . . . 14
6.2.4 Memory Management: Record's Metadata Overhead . . . 14
6.3 Ingestion and Archival Optimizations: Two Log-structured Techniques . . . 15
6.3.1 First Technique: Active Uniform Segments. . . . 15
6.3.2 Second Technique: Replicate Group of Segments Similarly. . . . 16
6.3.3 Other Considerations. . . . 16
6.4 HDFS Extension: A New Block Placement Strategy . . . 17
6.5 The DIPS Interface: Required Stream Metadata for Efficient Processing . . . 17
6.5.1 Management of Producers. . . . 17
6.5.2 Management of Consumers. . . . 18

7 Strengths and Limitations of State of Art Ingestion and Storage Systems **19**

8 Discussion **20**

9 Conclusion and Next Steps **21**



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803